

CMSC 714
Lecture 3
Message Passing with
MPI

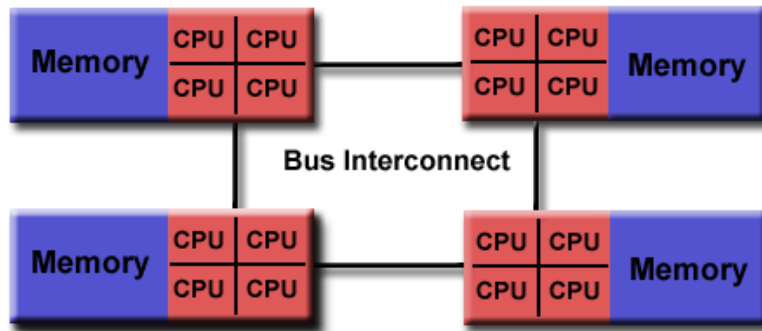
Alan Sussman

Notes

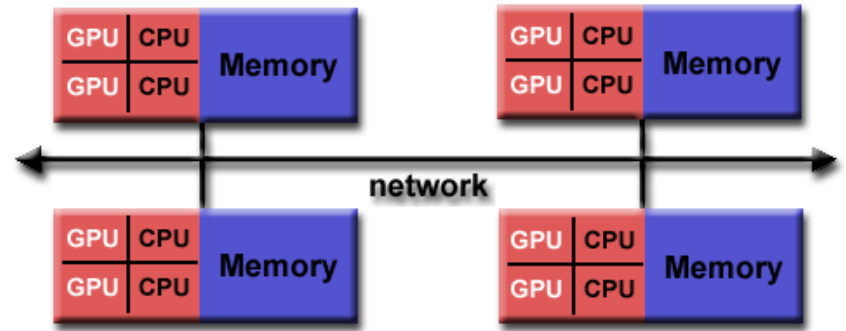
- To access papers in ACM or IEEE digital library, must come from a UMD IP address
- Login info for zaratan cluster will be provided Thursday, used for all assignments
- First assignment (MPI) announced by end of this week or early next week
- Check Readings page to see when you are assigned to send questions for a lecture
 - Starts for next week's lectures
 - 3-4 questions on average, more is OK
 - by 6PM day before lecture

Distributed memory architecture

- Each processor/core only has access to its local memory
- Writes in one processor's memory have no effect on another processor's memory



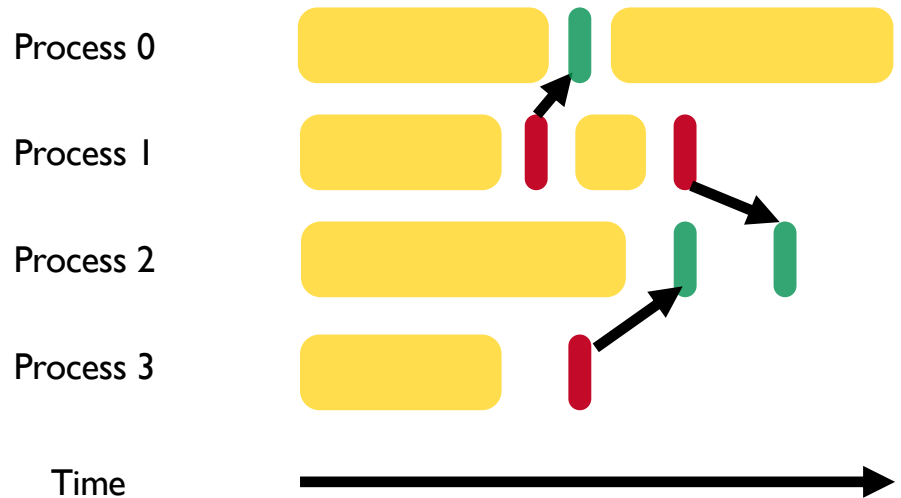
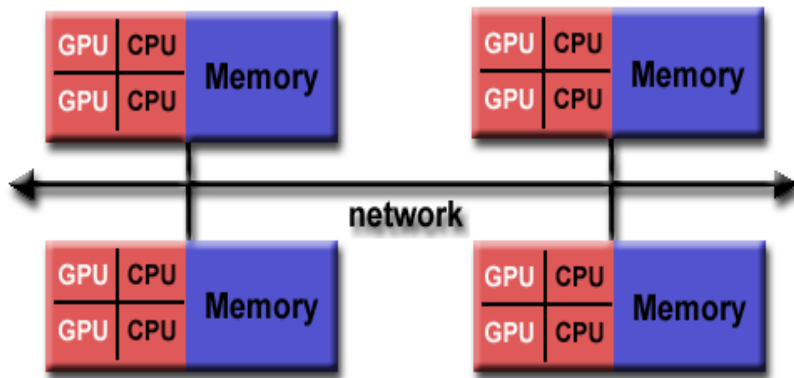
Non-uniform Memory Access (NUMA)



Distributed memory

Distributed memory programming models

- Each process only has access to its own local memory / address space
- When processes need data from remote processes, they have to send/receive messages



Message passing

- Parallel programming model
 - Parallelism is achieved by making calls to a library and the execution model depends on the library used.
- Parallel runtime system:
 - Implements the parallel execution model
- A parallel message passing program consists of independent processes
 - Processes created by a launch/run script
- Each process typically runs the same executable, but potentially different parts of the program
- Often used for SPMD style of programming

MPI

- **Goals:**

- Standardize prior message passing designs/implementations:
 - PVM, P4, NX (Intel), MPL (IBM), ...
- Support copy-free message passing
- Portable to many platforms – *defines an API, not an implementation*

- **Features:**

- point-to-point messaging
- group/collective communications
- profiling interface: every function has a name-shifted version

- **Buffering (in standard mode)**

- no guarantee that there are buffers
- possible that send will block until receive is called

- **Delivery Order**

- two sends from same process to same dest. will arrive in order
- no guarantee of fairness between processes on receive

Hello World in MPI

```
#include "mpi.h"
#include <stdio.h>

int main(int argc, char *argv[]) {
    int rank, size;
    MPI_Init(&argc, &argv);

    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    printf("Hello world! I'm %d of %d\n", rank, size);

    MPI_Finalize();
    return 0;
}
```

Compiling and running an MPI program

- Compiling:

```
mpicc -o hello hello.c
```

- Running:

```
mpirun -n 2 ./hello
```


Process creation / destruction

- `int MPI_Init(int argc, char **argv)`
 - Initialize the MPI execution environment
- `int MPI_Finalize(void)`
 - Terminates MPI execution environment

MPI Communicators

- Provide a named set of processes for communication
 - plus a *context* – system allocated unique tag
- All processes within a communicator can be named
 - a communicator is a group of processes and a context
 - numbered from 0...n-1
- Allows libraries to be constructed
 - application creates communicators
 - library uses it
 - prevents problems with posting wildcard receives
 - adds a communicator scope to each receive
- All programs start with `MPI_COMM_WORLD`
 - Functions for creating communicators from other communicators (split, duplicate, etc.)
 - Functions for finding out about processes within communicator (size, my_rank, ...)

Process identification

- `int MPI_Comm_size (MPI_Comm comm, int *size)`
 - Determines the size of the group associated with a communicator
- `int MPI_Comm_rank (MPI_Comm comm, int *rank)`
 - Determines the rank (ID) of the calling process in the communicator

Send a message

```
int MPI_Send(const void *buf, int count, MPI_Datatype  
datatype, int dest, int tag, MPI_Comm comm)
```

buf: address of send buffer

count: number of elements in send buffer

datatype: datatype of each send buffer element

dest: rank of destination process

tag: message tag

comm: communicator

Receive a message

```
int MPI_Recv( void *buf, int count, MPI_Datatype datatype,  
int source, int tag, MPI_Comm comm, MPI_Status *status )
```

buf: address of receive buffer

status: status object

count: maximum number of elements in receive buffer

datatype: datatype of each receive buffer element

source: rank of source process

tag: message tag

comm: communicator

Simple send/receive in MPI

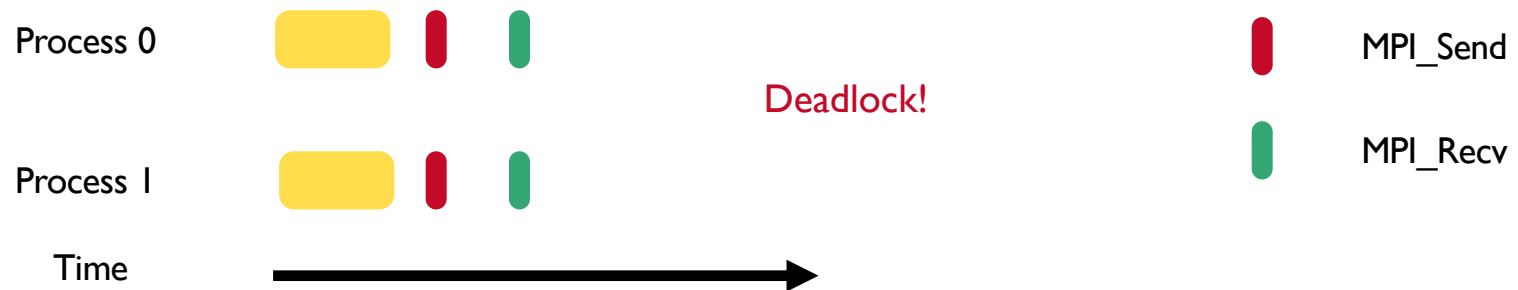
```
int main(int argc, char *argv) {
    ...
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    int data;
    if (rank == 0) {
        data = 7;
        MPI_Send(&data, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
    } else if (rank == 1) {
        MPI_Recv(&data, 1, MPI_INT, 0, 0, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
        printf("Process 1 received data %d from process 0\n",
data);
    }

    ...
}
```

Basic MPI_Send and MPI_Recv

- MPI_Send and MPI_Recv routines are blocking
 - Only return when the buffer specified in the call can be used
 - Send: Returns once sender can reuse the buffer
 - Recv: Returns once data from Recv is available in the buffer



Non-Blocking Point-to-point Functions

- Two Parts

- post the operation
- wait for results

- Also includes a poll/test option

- checks if the operation has finished

- Semantics

- must not alter buffer while operation is pending (wait returns or test returns true)
- and data not valid for a receive until operation completes

Collective Communication

- Communicator specifies process group to participate
- Various operations, that may be optimized in an MPI implementation
 - Barrier synchronization
 - Broadcast
 - Gather/scatter (with one destination, or all in group)
 - Reduction operations – predefined and user-defined
 - Also with one destination or all in group
 - Scan – prefix reductions
- Collective operations may or may not synchronize
 - Up to the implementation, so application can't make assumptions

MPI Calls

- Include `<mpi.h>` in your C/C++ program
- First call `MPI_Init(&argc, &argv)`
- `MPI_Wtime()`
 - Returns wall time
- At the end, call `MPI_Finalize()`
 - No MPI calls allowed after this

MPI Communication

- Parameters of various calls (in later example)
 - var – a variable (pointer to memory)
 - num – number of elements in the variable to use
 - type {MPI_INT, MPI_REAL, MPI_BYTE, ...}
 - root – rank of process at root of collective operation
 - src/dest – rank of source/destination process
 - status - variable of type MPI_Status;
- Calls (all return a code – check for MPI_Success)
 - MPI_Send(var, num, type, dest, tag, MPI_COMM_WORLD)
 - MPI_Recv(var, num, type, src, MPI_ANY_TAG, MPI_COMM_WORLD, &status)
 - MPI_Bcast(var, num, type, root, MPI_COMM_WORLD)
 - MPI_Barrier(MPI_COMM_WORLD)

MPI datatypes

- All messages are typed
 - base/primitive types are pre-defined:
 - int, double, real, {unsigned}{short, char, long}
 - MPI_INT, MPI_DOUBLE, MPI_CHAR, ...
- Derived or user-defined datatypes:
 - Array of elements of another datatype
 - struct data type to accommodate sending multiple datatypes

MPI Misc.

- **Processor Topologies**
 - Allows construction of Cartesian & arbitrary graphs
 - May make it easier to map processes to processors/nodes for some communication patterns
 - May allow some systems to run faster
- **Language bindings for C, Fortran, C++, ...**
- **What else is in current versions of MPI**
 - Dynamic process creation
 - Parallel I/O – MPI-IO
 - One-sided communication
 - And much more – current version of standard is MPI 4, committee is working on MPI 5
 - See <https://www.mpi-forum.org/>

Sample MPI Program

```
#include "mpi.h"

int main(int argc, char **argv) {
    int myrank, friendRank;
    char message[MESSAGESIZE];
    int i, tag=MSG_TAG;
    MPI_Status status;

    /* Initialize, no spawning necessary */
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
    if (myrank==0) { /* I am the first process */
        friendRank = 1;
    }
    else { /* I am the second process */
        friendRank=0;
    }
    MPI_Barrier(MPI_COMM_WORLD);
    if (myrank==0) {
        /* Initialize the message */
        for (i=0 ; i<MESSAGESIZE ; i++) {
            message[i]='1';
        }
    }

    /* Now start passing the message back and forth */
    for (i=0 ; i<ITERATIONS ; i++) {
        if (myrank==0) {
            MPI_Send(message, MESSAGESIZE,
                MPI_CHAR, friendRank, tag,
                MPI_COMM_WORLD);
            MPI_Recv(message, MESSAGESIZE,
                MPI_CHAR, friendRank, tag,
                MPI_COMM_WORLD, &status);
        }
        else {
            MPI_Recv(message, MESSAGESIZE,
                MPI_CHAR, friendRank, tag,
                MPI_COMM_WORLD, &status);
            MPI_Send(message, MESSAGESIZE,
                MPI_CHAR, friendRank, tag,
                MPI_COMM_WORLD);
        }
    }
    MPI_Finalize();
    exit(0);
}
```

For more details

- <https://www.mpi-forum.org>
 - includes 4.1 documentation (API), but goes all the way back to 1.0
 - 5.0 under development
- books from MIT Press include *Using MPI* and *MPI: The Complete Reference*
- multiple public domain implementations available
 - mpich2 – Argonne National Lab and open source team – <https://www.mpich.org/>
 - OpenMPI – large open source team – <https://www.open-mpi.org>
 - MVAPICH – high performance implementation from OSU - <https://mvapich.cse.ohio-state.edu/>
 - vendor implementations available too (Intel, IBM, HPE/Cray, ...)
- for zaratan cluster info, see <https://hpcc.umd.edu/hpcc/help/usage.html>